Exceptions et Asynchronisme

Guillaume Chazarain

Rapport de stage de fin d'études

Filière SAR/Master STIC Spécialité RSD 2005

Projet OASIS (Projet commun INRIA, CNRS-I3S, UNSA)

Maître de stage : Denis Caromel

1er mars - 30 septembre 2005

Polytech'Nice-Sophia
Département Sciences Informatiques

École Polytechnique de l'Université de Nice - Sophia Antipolis
Département Sciences Informatiques
930 Route des Colles - BP 145 - 06903 Sophia Antipolis Cedex
Tél +33-4-92 96 50 50 - Fax +33-4-92 96 50 55
URL http://www.polytech.unice.fr/informatique

Remerciements

Je tiens tout d'abord à remercier Denis Caromel, mon encadrant, qui m'a guidé tout au long de mon stage. Il a su trouver des problèmes intéressants et me guider dans leurs études.

L'équipe OASIS au complet est aussi à remercier pour son accueil, avec une mention spéciale pour Christian Delbé qui m'a aidé durant toutes les étapes de mon travail, et pour Alexandre di Costanzo qui a partagé son bureau avec moi.

Je souhaite aussi remercier les personnes ayant relu ce rapport pour leurs contributions à son aboutissement.

Table des matières

Liste des figures

1 Introduction

La gestion des erreurs dans les programmes informatiques a toujours été un aspect primordial pour aboutir à des systèmes robustes. Les systèmes distribués, de plus en plus répandus, exacerbent cet aspect sur deux points. Premièrement, les occasions d'erreur croissent avec l'augmentation du nombre de ressources, et deuxièmement ces erreurs sont plus délicates à traiter puisque leur origine peut être diverse. Avec l'importance grandissante des grilles de calcul, cette gestion d'erreurs devient donc encore plus indispensable.

Une des techniques plébiscitées de gestion d'erreur (les exceptions) est basée sur un flot de contrôle synchrone, et ce flot est dérouté en cas d'erreur. Il serait intéressant de pouvoir utiliser cette technique de gestion d'erreurs couplée avec l'asynchronisme. Par exemple, les techniques de plus en plus répandues de grilles de calcul nécessitent un traitement d'erreurs dans un environnement asynchrone. Le principe du code asynchrone est de ne pas attendre la fin de l'exécution des instructions. En conséquence, du point de vue des exceptions, il se peut que le flot d'exécution doive être dérouté vers un endroit pour traiter une erreur, mais que cet endroit n'existe plus en raison de l'asynchronisme.

L'objectif de mon stage de recherche au sein de l'équipe OASIS à l'INRIA Sophia Antipolis était donc : adapter la gestion des exceptions à un environnement asynchrone. Nous détaillerons précisément par la suite ce que nous entendons par exceptions et asynchronisme. Nous explorerons les différentes solutions proposées dans la littérature et proposerons la nôtre. Elle sera finalement validée par un exemple d'application.

Pour résumer succinctement notre solution, elle se base sur le découpage en blocs des programmes. Ce découpage est imposé par le mécanisme d'exceptions. L'idée est de considérer ces blocs comme des instructions synchrones, ainsi on permet de l'asynchronisme au sein des blocs mais en considérant les blocs dans leur ensemble nous nous retrouvons dans un environnement synchrone. Dans cet environnement synchrone, nous pouvons utiliser le mécanisme des exceptions.

Ce travail [1] a été présenté au workshop[1] sur les exceptions de la conférence ECOOP[2] 2005 à Glasgow le 25 juillet.

[1] Workshop on Exception Handling in Object Oriented Systems
[2] European Conference on Object-Oriented Programming

2 Gestion d'erreurs

2.1 Traitement primitif d'erreurs

De nombreuses techniques de traitement d'erreurs sont utilisées couramment en programmation. Ce sont des techniques qui ne nécessitent aucun support particulier de la part du langage de programmation. Ces techniques sont encore utilisées avec des langages comme C car ils ne proposent pas d'autres gestion d'erreurs, il faut donc utiliser ces solutions « ad hoc ». De plus, il est intéressant de conserver à l'esprit les qualités et défauts de ces techniques afin de profiter de leur expérience. Nous allons donc décrire ces principales techniques.

2.1.1 Reroutage bas niveau

Description

Ce mécanisme est principalement associé au langage C et propose deux fonctions (en fait 4 pour des raisons techniques) : `setjmp()` et `longjmp()`. Le rôle de `setjmp()` est de sauvegarder dans un espace passé en paramètre le contexte d'exécution courant : généralement l'ensemble des registres, ce qui inclut un pointeur d'instruction courante et un pointeur de pile. Quant à `longjmp()`, son rôle est de restaurer le contexte passé en paramètre, c'est-à-dire qu'après cet appel le flux d'exécution se retrouve exactement après l'appel à `setjmp()`. Le code de retour de `setjmp()` permet de faire la différence entre un retour normal de l'appel et un retour à cause d'un `longjmp()`. La fonction `setjmp()` est donc utilisée pour sauvegarder un état correct alors que `longjmp()` est appelée en cas d'erreur pour revenir à cet état correct.

Critique

Sur l'exemple simpliste d'utilisation en figure 2.1 une première limitation du mécanisme apparaît : le contexte est stocké dans une variable (ici `jmp_buf env`). Une difficulté se présente donc : passer cette variable de l'appelant de `setjmp()` à l'appelant de `longjmp()`.

Dans cet exemple, la fonction `dangerous_call()` est appelée, et si elle détecte un problème elle effectue une sorte de « backtrack » au dernier état correct connu. Examinons de plus près l'exécution de cet exemple :

1. cette variable est utilisée pour stocker le « contexte » du flot d'exécution,
2. le premier appel à `setjmp` sauvegarde le contexte et retourne 0,
3. ici la fonction se contente de déclencher le mécanisme d'erreur,
4. cette ligne ne sera jamais atteinte puisque le `longjmp` a dérouté le flot d'exécution,
5. le flot a été amené au `setjmp` correspondant qui renvoie la valeur 1 passée à `longjmp()`,
6. on se trouve donc dans l'autre branche du `if` et le programme peut terminer dans de bonnes conditions.

```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf env; /* 1. */

static void dangerous_call()
{
    /*
     * ...
     * if (error)
     */
    longjmp(env, 1); /* 3. */
}

int main(void)
{
    if (setjmp(env) == 0) { /* 2. */ /* 5. */
        /* Première fois, on lance l'appel */
        dangerous_call();
        puts("Ok"); /* 4. */
        return 0;
    } else {
        puts("Échec"); /* 6. */
        /* L'exécution peut se poursuivre normalement ... */
        return 1;
    }
}
```

Figure 2.1. Exemple simple d'utilisation de `setjmp()/longjmp()`

Même s'il n'apparaît pas dans des exemples simplistes, il y a un autre problème que celui du passage de paramètre entre `setjmp()` et `longjmp()` : la complexité ajoutée. Ces appels de bas niveau rendent les programmes instables, ce qui va à l'encontre de leur but : gérer les erreurs. La principale source d'instabilité est l'utilisation de `longjmp()` pour se retrouver dans une fonction qui a terminé son exécution. Ceci arrive quand la fonction qui appelle `setjmp()` se termine et qu'un appel `longjmp()` provoque un retour dans cette même fonction. La pile ayant été écrasée par les autres appels de fonction, les variables locales seront corrompues. Le schéma d'exécution pour cette situation est le suivant :

- une fonction `f()` est exécutée, elle crée des variables locales,
- elle fait appel à `setjmp()` et stocke le résultat dans `env`,
- cette fonction se termine,
- une fonction `g()` appel `longjmp()` avec le paramètre `env`,
- le flot de contrôle se retrouve dans le fonction `f()` alors qu'elle était terminée,
- la pile ne contient plus les variables de `f()`, le comportement est imprévisible.

Cet aspect est encore plus important quand le couple `setjmp()/longjmp()` est utilisé à tort comme solution de checkpointing. En effet, la figure 2.2 montre un exemple de ce genre d'application. Des « checkpoints » sont créés en appelant `sigsetjmp()` et on retourne automatiquement au dernier généré en cas d'erreur. La faute dans cette utilisation est de traiter à tort le signal de « segmentation fault » comme un signal ordinaire. En effet, si ce signal est déclenché, le programme est dans un état indéterminé, il est donc impossible de récupérer l'erreur de manière robuste.

```
#include <signal.h>
#include <stdio.h>
#include <setjmp.h>

static jmp_buf env;

static void dangerous_call()
{
    int *ptr = NULL;

    /* Déréférencement invalide d'un pointeur nul */
    *ptr = 0;
}

static void handle_segv(int sig)
{
    siglongjmp(env, 1);
}

int main(void)
{
    signal(SIGSEGV, handle_segv);

    if (sigsetjmp(env, 1) == 0) {
        /* Première fois, on lance l'appel */
        dangerous_call();
        puts("Ok");
        return 0;
    } else {
        puts("Échec");
        /* L'exécution peut se poursuivre normalement ... */
        return 1;
    }
}
```

Figure 2.2. Utilisation erronée de `setjmp()/longjmp()`

Ce mécanisme est sûrement le plus déroutant parmi ceux présentés car il donne l'impression de pouvoir récupérer n'importe quelle erreur.

On retiendra de cette solution qu'elle est trop générique, et qu'il est donc impossible de la spécifier uniquement pour une utilisation de gestion d'erreurs. De plus, son opération de bas niveau déstabilise l'application l'utilisant car les incohérences entre les registres et le reste de l'environnement provoquent des comportements imprévisibles.

2.1.2 Code de retour de fonction

Description

Une autre approche de gestion d'erreurs, continuellement utilisée même avec les langages offrant une stratégie intégrée d'exceptions, est de gérer une erreur comme une valeur de retour impossible. Ainsi on suppose que l'appelant vérifie la valeur qu'il reçoit afin de s'apercevoir du problème. Par exemple, un booléen faux ou une instance nulle peut indiquer une erreur. L'amélioration immédiate de cette solution est d'encoder dans ce code de retour des informations sur l'erreur, par exemple un nombre entier peut devenir un code d'erreur s'il est négatif. Cette solution est si populaire grâce à son principal avantage : sa simplicité.

Critique

Cette solution a deux problèmes principaux. Le premier est qu'elle n'est pas automatique, c'est-à-dire que si le programmeur oublie de vérifier la valeur renvoyée par un appel, l'application subira des dysfonctionnements. C'est donc une contrainte supplémentaire pour le programmeur.

Le deuxième problème est plus conceptuel, mais ces effets sont tout autant pratiques. Le type de retour d'une fonction peut être un type « erreur » si la seule utilité de cette valeur de retour est de signaler une erreur. Par exemple, une fonction pour envoyer un message peut avoir comme type de retour une description détaillée de l'erreur. Si ce n'est pas le cas, le type de retour normal sera « surchargé » avec des informations sur l'erreur et le résultat ne sera donc pas vraiment du type attendu. Par exemple, une fonction pour recevoir un message aura un type de retour susceptible de contenir soit un message soit une erreur. En pratique, cette limitation signifie qu'il n'est pas toujours possible de fournir de l'information sur une erreur dans un objet (au sens large) prévu pour contenir des données traitées par l'application en fonctionnement normal. Cette technique n'est donc pas applicable au cas général, mais uniquement au cas par cas.

L'enseignement à retenir de cette approche est la nécessité de séparer le flux de retour de valeurs standard du flux de retour d'erreurs (comme le fait Unix avec `stdout` et `stderr`).

2.1.3 État global d'erreur

Description

L'alternative principale à la solution précédente consiste à maintenir une variable globale contenant la dernière erreur. L'avantage de cette méthode est qu'elle résout le problème d'accès aux informations sur l'erreur car il est possible de fournir une API utilisable à tout moment permettant de consulter ces informations.

De plus, elle sépare bien le flux de retour normal avec le flux d'erreur car ce dernier est représenté par l'état global d'erreur. Cependant, la séparation est trop importante car le programmeur doit se rappeler de consulter cet état global après chaque appel utilisant ce mécanisme. En pratique, comme en C avec `errno` par exemple, cette solution est combinée avec la précédente, c'est-à-dire qu'un appel signal une erreur dans sa valeur de retour, et plus d'informations sur cette erreur sont obtenues en consultant l'état global.

Critique

Le principal problème de cette approche concerne la durée de vie de l'état global en question car il peut être modifié à tout moment. L'exactitude des informations contenues dans cet état est souvent assurée uniquement au retour d'un appel. En effet, il est possible qu'entre la mise à jour de l'état et sa consultation un autre appel échoue, et donc modifie la valeur de l'erreur. Ceci est encore plus vrai dans les programmes multi-threadés. Un palliatif souvent utilisé est de conserver une liste d'erreurs plutôt qu'une seule, mais il devient assez contraignant de travailler avec plusieurs erreurs en même temps.

2.2 Les exceptions

Après avoir vu ces techniques de gestion d'erreurs, intéressons-nous aux exceptions [2]. Nous détaillons cette notion dans cette section afin de pouvoir l'utiliser dans le reste du mémoire.

2.2.1 Définition

Une exception peut être définie comme un évènement exceptionnel. Plus précisément, ce critère exceptionnel doit exclusivement consister en une erreur. Théoriquement, durant une exécution normale d'un programme aucune exception ne doit survenir. Une exception est donc signalée lors d'une condition d'erreur dans un programme.

2.2.2 Traitement

Un autre aspect des exceptions est qu'après être signalées, elles sont censées être traitées. C'est pourquoi, en pratique il n'est pas rare qu'un programme émette des exceptions tout en conservant une exécution normale sans erreur visible par l'utilisateur.

Quand elles sont implémentées dans un langage de programmation, ce qui constitue leur principale utilisation, elles nécessitent de nouvelles constructions pour introduire leurs

sémantiques. De plus, l'exception elle-même est souvent réifiée afin d'être représentée dans le programme comme un « objet » de première classe.

Les constructions nécessaires pour un mécanisme d'exceptions sont des blocs de code de gestion d'erreurs, où l'exception comme objet de première classe est accessible. Ces blocs de gestion d'erreurs sont associés lexicalement au code normal mais aussi au type d'exception à traiter. Ainsi quand une exception est signalée, le flot de contrôle est redirigé vers le bloc correspondant. Voyons un exemple abstrait d'une telle construction en figure 2.3.

```
try {
        code normal exécuté // 1
        try {
                code normal exécuté // 2
                signale MonException // 3
                code normal ignoré
        } catch UneException {
                code de gestion d'erreur pour UneException, ignoré ici // 4
        } finally {
                code exécuté dans tous les cas // 5
        }
        code normal ignoré
} catch MonException {
        code de gestion d'erreur pour MonException, exécuté // 6
}
```

Figure 2.3. Exemple abstrait de construction utilisant les exceptions

Dans cet exemple abstrait, nous avons une imbrication de blocs de code « normal ». Lorsque l'exception est signalée, le flot de contrôle traverse l'imbrication pour se retrouver dans le bloc associé à l'exception puis continue l'exécution à partir de ce bloc. Le flot de contrôle sur l'exemple suit donc l'ordre croissant des nombres en annotation.

La fonctionnalité intéressante des exceptions est justement cette association qui est réalisée automatiquement soit dynamiquement soit statiquement si le mécanisme est suffisamment limité pour que ce soit possible. Quand une association statique est possible, le compilateur peut directement insérer les redirections vers le code de gestion d'erreurs approprié. En général, ce n'est pas le cas, et le bloc correspondant est découvert dynamiquement. Pour se faire, au fur et à mesure de l'exécution du code et de l'imbrication de blocs de code associés avec les blocs de gestion d'erreurs, une pile de gestionnaires d'exceptions est créée. Chaque niveau de cette pile contient une association entre un ou plusieurs types d'exceptions et des blocs de code gérant ces exceptions. Nous appelons cette pile la pile des masques d'exceptions.

Pour rediriger le flot vers le bloc de gestion d'erreur, il suffit donc de remonter cette pile en s'arrêtant au premier bloc concernant un masque compatible d'exceptions. Évidemment, ce mécanisme est assez coûteux.

2.2.3 Modèles d'exceptions

Les mécanismes d'exceptions peuvent se classer en deux catégories : ceux *avec reprise* et ceux *sans*. La différence entre ces deux catégories apparaît au moment de sortir du code de gestion d'exception. Le modèle avec reprise permet de recommencer une exécution qui a échoué, l'idée sous-jacente est qu'après l'échec on suppose que le problème a été réparé. Au contraire, le modèle sans reprise est plus pessimiste. Le but est plus de terminer dans de bonnes conditions que de retenter l'exécution. La pratique et la théorie s'accordent pour préférer l'approche sans reprise car sa sémantique, surtout vis-à-vis des effets de bords, est plus simple. De toutes façons, les deux modèles ne sont pas mutuellement exclusifs puisque l'on peut simuler l'un avec l'autre. Nous nous plaçons donc dans un contexte sans reprise, qui est celui proposé par tous les systèmes d'exceptions modernes.

2.2.4 Apports des exceptions

En reprenant la présentation des autres approches de gestion d'erreur, nous pouvons voir comment les exceptions corrigent leurs imperfections.

Par rapport au reroutage bas niveau (`setjmp()`/`longjmp()`) :

- possibilités de déroutement moindres, donc flot de contrôle plus prévisible,
- aucune transmission de contexte nécessaire.

Par rapport à la surcharge du code de retour des fonctions :

- notification automatique de l'erreur,
- séparation réelle de la valeur de retour de l'erreur,
- plus de possibilité pour décrire l'erreur car elle n'a pas à cohabiter avec la valeur de retour.

Par rapport à l'état global d'erreur :

- notification automatique de l'erreur,
- durée de vie de l'exception assurée.

3 Asynchronisme et Java

3.1 Programmation distribuée asynchrone

3.1.1 Programmation distribuée

La programmation distribuée consiste à répartir l'exécution d'une application sur plusieurs espaces d'adressage. Ceci s'étend d'une simple répartition sur plusieurs processus à une répartition sur plusieurs ordinateurs potentiellement dans des réseaux différents. La difficulté dans l'utilisation de plusieurs espaces d'adressage réside dans le fait qu'il n'y a plus d'état partagé, ainsi communications et synchronisation doivent être effectuées explicitement.

De nombreux projets ont pour but de masquer cette complexité et de fournir un environnement de programmation aussi complet que celui que l'on trouve en programmation ordinaire. Un des projets les plus en vue dans ce domaine est Java RMI[3] qui permet de programmer en Java habituel. RMI se charge de distribuer l'exécution des programmes.

Comme on peut le voir sur la figure 3.1, RMI s'occupe de sérialiser les appels de méthodes avec leurs arguments (marshalling), les envoyer au serveur, puis une fois que le serveur a traité l'appel, retourner par le même chemin le résultat au client.

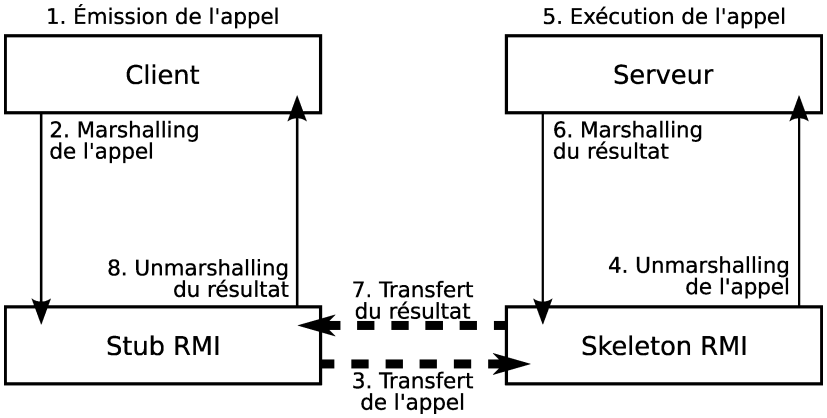


Figure 3.1. Modèle client-serveur de RMI

3.1.2 Asynchronisme transparent

Le modèle client-serveur de RMI que nous venons de voir a l'avantage de la simplicité. En contrepartie, ses performances ne sont pas optimales car il ne profite pas totalement de la distribution. En effet, pendant que le serveur exécute l'appel du client, ce dernier est mis en attente et ne sera réveillé qu'une fois le résultat revenu. Contrairement au cas non distribué, les ressources du client ne sont pas utilisées pour exécuter un appel de

[3] Remote Method Invocation

méthode, il est donc envisageable d'utiliser ces temps d'attente plus intelligemment du côté du client.

Une approche couramment utilisée pour pallier ce problème est la sémantique de l'envoi de message. Ceci, fourni par exemple par MPI [3], n'a pas pour but de simuler un environnement de programmation standard (RPC) mais au contraire permet d'optimiser une application distribuée en contrôlant précisément la partie distribution.

Quant à l'asynchronisme transparent, sa finalité est de proposer un compromis entre le synchronisme à la RMI et l'asynchronisme à la MPI. Comme RMI, il s'utilise en reprenant le paradigme de l'appel de méthode et la notation pointée, mais il se différencie sur le comportement une fois l'appel lancé. Là où RMI attend la réponse, avec un mécanisme d'asynchronisme transparent, l'appel retourne tout de suite afin de profiter du temps de calcul pour lancer un autre appel par exemple sur une autre machine ou continuer l'exécution en local.

Nous nous plaçons donc dans un contexte d'asynchronisme transparent car il permet de gagner en efficacité.

3.2 Environnement

Le mécanisme que nous proposons sera implémenté en utilisant le langage Java et s'intègrera à la bibliothèque ProActive [4], il convient donc de les présenter tous les deux.

3.2.1 Les exceptions en Java

L'objet de ce travail n'étant pas de décrire le langage Java, nous nous concentrons uniquement sur sa gestion des exceptions. Elle est basée sur cinq mots-clés : `try`, `catch`, `finally`, `throw` et `throws`, et est inspirée de celle de C++.

Les mots-clés `try`, `catch` et `finally` servent à définir syntaxiquement les différents blocs utiles au mécanisme d'exceptions. Le bloc `try` est celui qui introduit ce que nous avons appelé le code « normal », le code dans ce bloc n'a pas à se soucier de la gestion d'erreurs. Puis, se succèdent autant de blocs `catch` que nécessaires, chacun recevant une exception d'un type donné dans le but de la traiter. Ce sont donc les blocs de gestion d'erreurs. Le dernier bloc est optionnel, et il s'agit du bloc `finally`, le programmeur est assuré que le code placé dans ce bloc sera exécuté quelque soient les détournements de flot de contrôle produits par les exceptions. Le bloc `finally` est exécuté même si aucune exception n'est signalée.

Le flot de contrôle avec les exceptions dans le cas de Java devient le suivant : le bloc `try` est exécuté normalement, quand une exception est signalée via la primitive `throw` elle remonte la pile en abandonnant l'exécution des blocs `try` rencontrés. Pour chaque bloc `try` abandonné en remontant la pile, son potentiel bloc `finally` est exécuté. Cette remontée s'arrête lorsqu'un bloc `catch` correspondant au type de l'exception est trouvé, il est donc exécuté. Une exception telle que signalée par un `throw` ou reçue par un `catch`

est simplement un objet standard Java héritant de la classe `java.lang.Exception`, et pouvant donc être manipulé comme tous les autres objets. Les blocs `finally` juxtaposés aux blocs `try` sont exécutés même si aucune exception n'est signalée.

Java trouve dynamiquement le bloc `catch` correspondant à une exception signalée. Pour chaque `try` les types d'exceptions gérés sont connus en consultant la liste des blocs `catch` successifs, cette liste donne une sorte de « masque d'exceptions » qui sera utilisé dynamiquement pour savoir si une exception est gérée dans le bloc en cours où s'il faut remonter la pile. Ainsi parallèlement à la pile d'appels standard Java maintient une pile de masques d'exceptions, elle est incrémentée en entrant dans un bloc `try` de son masque associé, et est décrémentée en sortant du bloc. Quand une exception est signalée, la pile est remontée jusqu'au premier bloc `catch` pouvant traiter ce type d'exception.

Quant au mot-clé `throws`, il est utilisé uniquement à la compilation et permet d'assurer statiquement que toutes les exceptions signalées sont traitées par un bloc `catch`, même si l'on ne peut pas statiquement associer une exception à son bloc `catch`. Le compilateur ne peut pas prédire le trajet de l'exception, mais garantit que tous les trajets possibles aboutissent à un bloc `catch`. S'il ne peut pas le garantir, il signale une erreur. Ce mot-clé est donc utilisé dans les signatures de méthodes pour indiquer les exceptions pouvant être signalées suite à l'appel de la méthode. La vérification de la clause `throws` porte à la fois sur les exceptions lancées directement via l'instruction `throw` et sur les exceptions remontées par d'autres méthodes appelées en examinant les déclarations `throws` de chaque méthode appelée.

Ce mécanisme de vérification de la clause `throws` concerne toutes les exceptions sauf celles qui héritent de la classe `java.lang.RuntimeException`. Les exceptions de cette classe sont habituellement utilisées pour signaler des erreurs à l'exécution liées à des erreurs dans le code, comme par exemple des déréférencements de pointeurs nuls ou des débordements de tableaux. Elles ne peuvent donc pas être incluses dans une déclaration `throws` vu que la localisation de ces erreurs est a priori inconnue, et potentiellement partout. Voyons un exemple d'utilisation de ce système d'exceptions en figure 3.2

La trace d'exécution est :

1. l'exception est signalée,
2. ce code n'est pas exécuté car l'exception a provoqué la sortie de la méthode,
3. l'exception n'est pas une `RuntimeException` qui peut arriver à tout moment, donc ce code n'est pas exécuté,
4. ce code est exécuté car il annonce le bon type d'exception,
5. ce code est exécuté aussi, même si aucune exception n'avait été signalée.

3.2.2 ProActive

La bibliothèque ProActive, à laquelle s'intégrera le mécanisme d'exceptions proposé, est une bibliothèque de programmation parallèle, concurrente et distribuée, c'est un des

```
class DangerousException extends Exception {}

public class A {

    private static void dangerousMethod() throws DangerousException {
       if (0 + 0 == 0) {
           throw new DangerousException(); // 1.
       }

        System.out.println("ici"); // 2.
    }

    public static void main(String[] args) {
       try {
           dangerousMethod();
        } catch (RuntimeException re) {
           System.out.println("Unexpected exception: " + re); // 3.
       } catch (DangerousException e) {
           System.out.println(e); // 4.
       } finally {
           System.out.println("All's well that ends well"); // 5.
       }
    }
}
```

Figure 3.2. Exemple d'utilisation des exceptions en Java

résultats du projet de recherche OASIS. Nous nous intéresserons uniquement à ses fonctionnalités nécessaires à la compréhension du mécanisme de traitement d'exceptions proposé.

ProActive repose sur un modèle MIMD (Multiple Instructions, Multiple Data) et sur la notion d'objet actif, l'objet possède une activité propre.

Le modèle de ProActive présente les caractéristiques suivantes :

- des objets actifs et accessibles à distance,
- la séquentialité des activités (processus purement séquentiels),
- une communication par appels de méthode standard entre objets actifs,
- des appels asynchrones vers les objets actifs,
- un mécanisme d'attente par nécessité (futur transparent),
- les continuations automatiques (un mécanisme transparent de délégation),
- l'absence d'objets partagés,
- une programmation des activités qui est centralisée et explicite par défaut,
- du polymorphisme entre objets standard et objets actifs distants.

En l'absence d'extension syntaxique les programmeurs écrivent un code standard avec ProActive. La bibliothèque est elle-même extensible par les programmeurs, la conception du système est ouverte pour des adaptations et des optimisations.

Les objets actifs reçoivent des requêtes qu'ils exécutent séquentiellement. Vis-à-vis du client qui envoie la requête, l'exécution peut être asynchrone ou non selon différents facteurs. Nous entendons asynchrone, au sens qu'un appel de méthode est écrit normalement, mais au lieu d'attendre la fin de cet appel, l'exécution du code appelant se poursuit. Un appel sera asynchrone si toutes les conditions suivantes sont vérifiées :

- le type de retour est une classe héritant de `java.lang.Object` (quasiment tout à part les types primitifs et les tableaux) et implémentant l'interface `Serializable`, ou `void`,
- le type de retour n'est pas une classe `final` (écarte les `String`, `Integer`, ...),
- la méthode appelée n'a pas de `throws` dans sa signature.

La dernière condition a pour effet de rendre synchrone tous les appels à des méthodes susceptibles de signaler des exceptions. Notre travail a donc pour but, entre autres, de lever cette limitation.

Intéressons-nous maintenant de plus près aux appels asynchrones. Quand un de ces appels est déclenché, l'appelant reçoit immédiatement en retour un objet que nous appelons « futur ». Ce futur sera remplacé par le véritable résultat lorsque ce dernier sera connu. En attendant, c'est un objet présentant les mêmes caractéristiques que le résultat attendu à une différence près. Cette différence réside dans le comportement des appels sur cet objet : il n'est pas le véritable résultat attendu, son comportement quand il est sollicité est de se mettre en attente du résultat final. Cette attente est appelée « attente par nécessité ».

On a vu qu'il était aussi possible d'obtenir des appels asynchrones quand le type de retour de la méthode appelée est `void`. Dans ce cas, il est impossible de retourner un futur à l'appelant vu qu'il n'attend rien. Les appels de ce type sont qualifiés de « one-way » puisque l'appelant n'attend aucun retour.

La gestion des exceptions pour ces appels asynchrones sera détaillée dans l'état de l'art en section 4.2. Nous pouvons déjà signaler qu'une distinction entre les exceptions est maintenue au sein de ProActive. Celle-ci sépare les exceptions en « exceptions fonctionnelles » et « exceptions non-fonctionnelles ». Une exception fonctionnelle appartient au code « métier » du programmeur et concerne donc son fonctionnement, contrairement à l'exception non-fonctionnelle qui appartient au middleware. Ainsi, une exception fonctionnelle est signalée et traitée par le code de l'utilisateur. Une exception non-fonctionnelle est signalée par le middleware qui est donc libre de choisir un traitement pour cette exception.

La bibliothèque ProActive est programmée en gardant à l'esprit certaines contraintes afin de faciliter son utilisation par le programmeur final. Ces contraintes interdisent de modifier le bytecode des classes, de pré-traiter les sources avant compilation et de modifier la JVM. La librairie est donc écrite uniquement en pur Java. Ne pas respecter ces contraintes aurait pour effet de troubler le programmeur final car le code tel qu'il l'a écrit

ne serait pas celui qui est exécuté, ce qui rendrait le programme beaucoup plus difficile à déboguer. Par exemple, l'exécution ligne par ligne d'une méthode serait impossible si des instructions sont ajoutées dans le bytecode, désynchronisant l'état d'avancement dans le bytecode par rapport à celui dans le code source.

3.2.3 Problématique

Le but du mécanisme présenté est de gérer les exceptions signalées par des appels asynchrones. La difficulté réside dans le fait que les exceptions (ou plutôt leur absence) avec leurs blocs `try/catch` matérialisent la succession d'instructions sans erreur. Dans le cas asynchrone, nous supprimons cette succession d'instructions afin de créer du parallélisme, d'où le conflit. De plus, l'asynchronisme étant transparent par certains aspects, la gestion des exceptions asynchrone doit être la moins visible possible pour rester dans la philosophie.

Cette problématique intervient surtout pour les exceptions fonctionnelles, donc connues par le programmeur. Pour les exceptions non-fonctionnelles, nous avons une plus grande liberté d'action car elles ne sont pas directement liées au code du programmeur. Elles sont créées par le middleware qui est donc libre de les présenter au programmeur sous différentes formes.

4 État de l'Art

Dans la bibliothèque ProActive, les exceptions fonctionnelles sont séparées des exceptions non-fonctionnelles. À part ProActive, seul RMI fait cette différentiation, dans une moindre mesure. L'état de l'art n'en tiendra donc pas compte.

4.1 Exceptions non-fonctionnelles avec RMI

Le comportement le plus connu pour ces exceptions est celui de RMI avec ses `RemoteException` [5]. L'idée est qu'en passant l'appel de méthode par le réseau, on ajoute une possibilité d'erreur spécifique à cet endroit. RMI force donc le programmeur à ajouter l'exception `RemoteException` à la déclaration `throws` de chaque méthode exportée. En conséquence, les programmes utilisant RMI contiennent énormément de `throws RemoteException` et de blocs `try/catch` pour les `RemoteException`. À l'usage, cette gestion d'erreur omniprésente devient pesante et est réduite tout simplement à un bloc `catch (RemoteException re)` vide, c'est-à-dire que l'erreur est ignorée silencieusement.

Dans la plateforme .Net, venue après, ce problème a été traité en supprimant le mot-clé `throws`, c'est-à-dire que les méthodes ne peuvent plus indiquer leurs exceptions. Le compilateur ne vérifie plus le traitement de chaque exception potentielle. Cette alternative par rapport à Java est un sujet de discorde, et notre position est de préférer la version de Java à condition que l'erreur à la compilation soit remplacée par un avertissement. Ainsi, le programmeur est libre d'ignorer cet avertissement pour du code jouet qui n'a pas à être robuste, ou alors de s'aider de ces avertissements pour rendre son code plus robuste. Malheureusement, cette alternative n'existe pas à notre connaissance.

4.2 Gestion d'erreurs dans ProActive

Pour les exceptions normales (figurant dans une déclaration `throws`), le traitement est simple : les appels sont synchrones, donc on se retrouve dans un cas standard de gestion d'exception. Par contre, pour les `RuntimeException`, la situation se complique puisqu'on ne peut pas les prévoir. Dans le cas d'un appel asynchrone avec un type de retour, l'exception est mise dans le futur pour être signalée au premier accès au futur. Le problème de cette approche est que l'on risque de signaler l'exception en dehors d'un potentiel bloc `try/catch` lui étant destiné. Si l'appel est one-way, c'est-à-dire que le type de retour est `void`, une telle méthode n'est pas envisageable car il n'y a pas de futur dans lequel stocker l'exception. Le traitement dans ce cas, est assez primaire. Il consiste à afficher une trace de la pile du côté du serveur ; cette simplicité est imposée par le comportement « one-way » qui n'attend pas de réponse après l'envoi de la requête.

Lorsque la bibliothèque ProActive est utilisée au-dessus de RMI, elle utilise la `RemoteException` de RMI (ou plutôt son absence de déclenchement à l'exécution) pour

assurer que les requêtes sont bien envoyées, ainsi ProActive garantit le rendez-vous et la livraison de la requête à l'objet actif cible.

4.3 Fonctions de traitement

Cette approche consiste à associer de façon variée une fonction de traitement d'exception à une exception, et à appeler cette fonction quand l'exception associée sera signalée. Cette définition est volontairement générale car les multiples implémentations sont très variées.

Contrairement aux `RemoteException` qui s'intègrent avec le mécanisme d'exception du langage hôte, l'approche avec fonctions de traitement y est totalement orthogonale, et est donc indépendante du langage utilisé. En effet, les exceptions ne sont pas fournies par tous les langages, contrairement aux fonctions, du moins pour les langages utilisés en programmation distribuée.

Cette solution est la plus employée et après l'avoir présentée sous de multiples incarnations nous en verrons ses avantages et inconvénients.

4.3.1 Non-Functional Exceptions

Nous avons vu la gestion des exceptions fonctionnelles dans ProActive en section 4.2. Pour les exceptions non-fonctionnelles, ProActive propose un mécanisme appelé NFE [6] (Non-Functional Exceptions) qui consiste en une classification de ces exceptions et en l'association de ces exceptions à des fonctions de traitement à différents niveaux de priorités. Ces fonctions s'exécutent du côté client ou serveur en fonction de l'exception à traiter. Ce mécanisme permet d'ajouter des comportements intéressants comme un mode déconnecté pour les environnements mobiles, où les pannes provenant d'une déconnexion conduisent à une mise en attente des requêtes afin d'être relancées une fois reconnecté.

Ce mécanisme étant implémenté en Java, il ne se contente pas d'utiliser des fonctions mais plutôt des objets. En plus des fonctions nous avons donc un contexte et un état associés à ces objets, mais nous continuerons à les appeler fonctions car c'est leur aspect principal. Les fonctions de traitement répondent à l'interface de la figure 4.1 qui permet donc de savoir si une fonction s'applique, puis de l'appliquer. Les fonctions construites de cette façon sont dynamiquement associables aux exceptions correspondantes et à diverses entités.

```
public interface Handler {
    // Est-ce que cette fonction de traitement gère cette exception ?
    public boolean isHandling(NonFunctionalException e);

    // Traite l'exception
    public void handle(NonFunctionalException e);
}
```

Figure 4.1. Interface pour les fonctions de traitement NFE

4.3.2 Langage JR

Le langage de programmation JR [7] est une extension de Java fournissant un modèle de programmation concurrente basé sur le langage SR [8]. Les principales fonctionnalités ajoutées de JR sont :

- création dynamique de machine virtuelle,
- création dynamique d'objet distant,
- appels distants,
- communications asynchrones,
- sémantique de rendez-vous,
- création dynamique de processus.

Pour être compilés, les programmes écrits en JR sont préalablement transformés en Java, puis compilés normalement. Cette phase de transformation implique que le programme sera difficilement débogable puisque le code exécuté et analysé sera le code en Java et non celui en JR.

L'extension qui nous intéresse ici est le mot-clé `handler` [9]. Il est utilisé conjointement avec le mot-clé `send`, ce dernier servant à effectuer des appels asynchrones. Avec ces deux mots-clés, une fonction de traitement est associée avec l'appel asynchrone à l'exécution. Comme nous sommes dans un environnement orienté objet, ce n'est pas seulement une fonction qui est associée à cet appel, mais un objet complet avec toutes les informations annexes qu'il peut contenir. Le type de cet objet doit être une classe implémentant toutes les méthodes associées aux exceptions déclarées dans le `throws` de la méthode appelée. La figure 4.2 montre comment définir un handler, et la figure 4.3 montre comment l'associer à un appel asynchrone.

```
class IOHandler implements edu.ucdavis.jr.Handler {
    public handler void method(java.io.EOFException e) {
        System.out.println("Caught java.io.EOFException " +
                           "through IOHandler object");
    }

    public handler void method(java.io.FileNotFoundException e) {
        System.out.println("Caught java.io.FileNotFoundException " +
                           "through IOHandler object");
    }
}
```

Figure 4.2. Exemple de définition d'un handler en JR

On peut remarquer dans la définition du handler, que le choix de la fonction à utiliser est laissé à Java grâce à la surcharge. Ainsi, la liste de définitions de méthodes est très similaire à une série de blocs `catch`, du moins syntaxiquement.

```
IOHandler myHandler = new MyHandler();
...
send readFile("/path/to/my/file") handler myHandler;
...
```

Figure 4.3. Exemple d'utilisation du mot-clé `handler` en JR

L'annotation `handler` à côté des appels par `send` est obligatoire en JR pour chaque appel asynchrone de méthode ayant une liste `throws` non vide. Ceci est vérifié statiquement par le compilateur JR, il vérifie également que le handler traite tous les types d'exceptions potentiellement signalées.

L'inconvénient de ce mécanisme est que l'association de la fonction de traitement à l'appel asynchrone n'est pas transparente, c'est-à-dire qu'il revient au programmeur d'annoter chacun des appels asynchrones. En fait, ceci n'est pas une grande faiblesse puisque les appels asynchrones sont de toutes façons explicitement introduits par le programmeur pour bénéficier de la sémantique d'envoi de message. L'asynchronisme ici ne sert pas d'optimisation transparente.

4.3.3 Kava

Kava [10] est un protocole méta-objet utilisé pour appliquer un mécanisme de sécurité sur le code compilé Java. Le bytecode est transformé au chargement afin d'y insérer des vérifications de sécurité.

Ce projet a été utilisé pour déferrer le traitement des exceptions à une fonction annexe [11], et ce travail pourrait très facilement être intégré dans un environnement asynchrone pour gérer les exceptions.

Comme on peut le voir sur la figure 4.4 la fonction `afterMethodExecution()` fournie par Kava peut-être utilisée comme fonction de traitement si l'appel s'est terminé par une exception. L'argument `context` est une réification de l'appel de méthode traité, ce qui permet d'avoir plus d'informations que juste l'exception comme dans le cas de JR.

```
public class ExceptionHandler extends MetaObject {
    public void afterMethodExecution(IMethodExecution context) {
        if (context.isExceptionRaised()) {
            System.out.println("Exception " + context.getException() +
                               " was raised");
        }
    }
}
```

Figure 4.4. Exemple de traitement d'exception avec Kava

En Kava, un méta-objet (instance de `ExceptionHandler` par exemple ici) est associé à un objet normal via un fichier de configuration définissant les associations. Ce moyen d'association semble plus optimal que le mot-clé `handler` vu précédemment car la granularité des associations est paramétrable. Par contre, le reproche que l'on peut faire à

cette approche est que le recours à un fichier de configuration annexe peut engendrer des désynchronisations avec le reste du code. Par exemple, si le code est modifié mais pas le fichier de configuration, des incohérences peuvent apparaître.

4.3.4 ARMI

Le projet ARMI [12] ne s'intéresse pas uniquement au traitement des exceptions mais a pour but de fournir un mécanisme d'appels distants asynchrones en Java. L'asynchronisme est réalisé grâce à des stubs multi-threadés. Pour gérer les exceptions ils proposent deux mécanismes dont un utilisant des fonctions de traitement.

Comme dans ProActive, l'asynchronisme est rendu transparent au programmeur via l'utilisation d'objets « futurs ».

Le mécanisme de gestion d'exceptions avec fonctions de traitement est utilisé en associant au futur des couples <type d'exception, fonction de traitement>.

Ce projet n'ayant pas fourni d'implémentation, il n'est pas possible d'obtenir des informations précises à son propos.

4.3.5 Fermetures dans les langages fonctionnels

Avec les exceptions, nous avons un problème de capture de contexte, problème géré par les fermetures comme proposées dans les langages de programmation fonctionnels.

Une fermeture peut être assimilée à l'association d'une fonction et d'une capture de l'environnement obtenue au moment de la création de la fermeture. Un exemple de fonction de traitement définie comme une fermeture est montré en figure 4.5.

Traçons l'exécution de cet exemple :

1. nous initialisons notre variable à 0 (`some-value` = 0),
2. la fermeture capture le contexte y compris la variable `some-value`,
3. la variable `some-value` est modifiée dans le code normal (`some-value` = 1),
4. le substitut de bloc `try` est appelé en lui passant la fermeture à utiliser en cas d'erreur,
5. l'environnement de ce bloc ne contient pas la variable `some-value`, cette instruction est donc commentée car elle est illégale,
6. la fermeture est exécutée,
7. elle accède en lecture/écriture à son environnement (`some-value` = 2),
8. cette modification est visible dans le code normal.

Comme on peut le voir, la capture du contexte est complète puisque les environnements du code normal et de la fermeture sont partagés et modifiables dans les deux cas. En figure 4.6 nous simulons cette capture en Java, avec quelques limitations importantes.

Le comportement du code Java est semblable à celui en Scheme, les annotations numérotées conviennent toujours. Il y a quand même un problème dans la version Java. Il est contenu dans la ligne `final int[] someValue = new int[1];`. En effet, Java

```
; Ceci est un équivalent au try en Java, la fonction de traitement à
; appeler en cas d'erreur est passée en paramètre.
; Évidemment, la fonction elle-même devrait exécuter plus de "code
; métier" que simplement appeler la fonction de traitement d'erreur.

(define (try-with-error-handler error-handler)
     ; (set! some-value (+ some-value 1)) ; 5.
     (error-handler)) ; 6.

; some-value est notre variable locale à laquelle nous aurons accès
; à la fois dans le code normal et dans le code de gestion d'erreur.
; Contrairement à Java, le code de gestion d'erreur a un accès en
; lecture et écriture à l'environnement capturé dans la fermeture.

(let* ([some-value 0] ; 1.
       [error-handler (lambda () ; 2.
                        (set! some-value (+ some-value 1)))]) ; 7.
     (set! some-value (+ some-value 1)) ; 3.
     (try-with-error-handler error-handler) ; 4.
     (display some-value)) ; 8.

=> 2
```

Figure 4.5. Exemple de fermeture en Scheme

ne permet de voir dans les fermetures uniquement les variables déclarées en `final`, c'est-à-dire en lecture seule. La justification donnée à ce choix de Java est la suivante : la capture du contexte est donnée par copie à la fermeture plutôt qu'en donnant une adresse. Ainsi afin d'éviter toute désynchronisation entre l'original et la copie, seules les variables en lecture seule sont acceptées.

Il y a deux justifications au passage de contexte par copie. La première est que la fermeture pourrait être exécutée par une autre thread, et ainsi il y aurait de la concurrence au niveau des variables locales, ce qui est contraire à Java. La seconde justification concerne la durée de vie de la fermeture. Il est possible que la fermeture soit encore référencée après que la méthode qui l'a créée soit terminée. La méthode étant terminée, son espace de pile est écrasé, et donc la fermeture aurait une vision corrompue des variables locales.

La deuxième raison peut être annulée en allouant si nécessaire les variables locales dans le tas et non la pile mais ceci viole une autre règle de Java : toutes les allocations dans le tas passent par un `new`.

Cette limitation des fermetures de Java peut être contournée [13] comme dans l'exemple en encapsulant les variables locales dans un tableau ou une classe, dont la valeur (l'adresse) sera constante. Malheureusement, cette contrainte rend inutilisable l'utilisation des fermetures en Java dans un cas général.

```
public class Fermeture {
    private static void tryWithErrorHandler(Runnable errorHandler) {
       // someValue[0]++; // 5.
       errorHandler.run(); // 6.
    }

    public static void main(String[] args) {
       final int[] someValue = new int[1];
       someValue[0] = 0; // 1.

       Runnable errorHandler = new Runnable(){ // 2.
           public void run() {
               someValue[0]++; // 7.
           }
       };

       someValue[0]++; // 3.
       tryWithErrorHandler(errorHandler); // 4.
       System.out.println(someValue[0]); // 8.
    }
}
```

Figure 4.6. Approximation de fermeture en Java

4.3.6 Conclusion sur les fonctions de traitement

La gestion d'exceptions en utilisant des fonctions de traitement est donc une approche très répandue. Un de ces avantages est sa simplicité, un autre est que cette méthode laisse à son concepteur l'impression de donner une liberté totale au programmeur pour choisir un comportement de gestion d'exception. En effet, le programmeur final peut spécifier n'importe quelle fonction à utiliser. En fait, ce sentiment de liberté est assez trompeur puisque l'usage principal des exceptions *est impossible à réaliser* avec une fonction. Cet usage principal est la remontée de la pile d'appel jusqu'à un certain point. Alors que les exceptions sont principalement utilisées pour remonter la pile d'appel, les fonctions de traitement ne permettent que d'augmenter cette pile en effectuant des appels de méthodes. Dans un environnement asynchrone il est difficile de remonter la pile d'appel car il se peut qu'elle ait totalement changé entre le début de l'appel et sa terminaison. Ceci est le principe même des appels asynchrones, ne pas attendre la fin de l'appel. Néanmoins, lorsque l'asynchronisme est utilisé comme optimisation transparente il est important de continuer à fournir la possibilité de remonter la pile vu que cette possibilité est considérée comme acquise dans un environnement synchrone.

Une autre faiblesse des fonctions de traitement est leur représentation du contexte. Dans un bloc `try/catch` normal, le code de gestion d'erreur est écrit juste à côté du code à protéger, il apparaît donc normal de pouvoir accéder à toutes les variables locales dans ce bloc de gestion d'erreur. Malheureusement, avec des fonctions de traitement en Java,

cette information est inaccessible. Nous pensons que cette information est pourtant très importante car c'est elle qui indique l'état de l'exécution.

On ne peut pas dire que les variables locales indiquent l'état d'avancement de l'exécution puisque l'on se place dans un environnement asynchrone, cependant on peut quand même prétendre que ces variables donnent une sorte de « justification » de l'exécution.

```
calcul(ExpressionBinaire expression) {
    try {
        int operande1 = operande2 = INDEFINI;
        operande1 = expression.getOperande1().evalue();
        operande2 = expression.getOperande2().evalue();
        int resultat = operation(expression.getOperateur(),
                                 operande1, operande2);
    } catch (Exception e) {
        /*
         * Le code ici a accès (entre autre) à operande1,
         * operande2 et expression.
         */
    }
}
```

Figure 4.7. Pseudo-code de calcul d'expression

Le pseudo-code de la figure 4.7 permet de comprendre cette justification. Si tous les appels sont asynchrones, ce n'est que dans un hypothétique bloc `catch` que l'on a accès à l'expression et aux valeurs potentiellement calculées des opérandes. Une fonction de traitement n'aurait pas accès à ces informations qui sont des variables locales.

Les implémentations modernes de système d'exceptions (comme Java) atténuent cette limitation en permettant de construire des exceptions comme des objets de première classe, donc en laissant la possibilité d'y mettre toute information jugée utile. Ainsi quelque soit l'environnement synchrone ou asynchrone, des informations aisément fournies au programmeur sont l'exception elle-même et tous ses attributs, ainsi que l'objet `this` qui a provoqué l'exception. Une possibilité d'obtenir les variables locales de la fonction signalant l'exception est de les mettre dans une nouvelle exception qui encapsule l'originale. Mais ceci implique que l'exception est accessible en même temps que les variables locales intéressantes, ce qui est justement ce que l'on ne sait pas faire avec les fonctions de traitement.

Les fonctions de traitement ont un avantage par rapport aux blocs `try/catch` habituels. Le fait qu'ils ne sont pas syntaxiquement liés au code qu'ils protègent permet d'envisager des comportements génériques comme par exemple repartir du dernier checkpoint connu ou recommencer l'appel qui a échoué. On peut donc implémenter des traitements d'exceptions avec reprise, à condition de maîtriser l'évolution du contexte, c'est-à-dire comprendre comment il va évoluer. De plus, on peut espérer dynamiquement modifier

les associations entre les appels et leurs fonctions de traitement si les associations sont dynamiques plutôt que syntaxiques.

4.4 Exception dans le futur

La plupart des projets proposant des appels de méthodes asynchrones transparents utilisent le concept de futur pour représenter le résultat en attente. Ce futur peut donc être utilisé pour signaler une exception potentielle.

4.4.1 ARMI

Nous avons déjà rencontré ce projet car il propose aussi une gestion d'erreurs en utilisant des fonctions de traitement. Il présente donc une deuxième stratégie, celle de mettre l'exception dans le futur à la place du résultat et de la signaler au moment de l'accès au futur. Il est précisé que ce mécanisme n'a pas encore été implémenté et que des problèmes d'implémentation à ce niveau sont attendus. En effet, un objet futur peut être utilisé bien après l'appel de la méthode qui l'a créé, c'est-à-dire en dehors de son bloc `try`. L'exception dans le futur pourrait donc être signalée à des endroits imprévus. Nous verrons en section 5.3.2 que leurs craintes sont tout à fait fondées.

4.4.2 Java 1.5

Dans sa version 1.5 (nom de code Tiger), Java implémente des objets futurs [14] explicites. La différence principale avec les autres approches rencontrées est que celle-ci ne cherche absolument pas à être transparente pour le programmeur. Ceci est certainement un défaut dans le cas de l'asynchronisme comme optimisation transparente, mais en forçant le programmeur à consulter explicitement le résultat dans le futur, il n'aura aucune surprise quant au lancement de l'exception. Une limitation de cette approche est que l'exception est encapsulée dans une `ExecutionException`, ainsi le programmeur ne pourra pas directement utiliser sa série de blocs `catch` pour aiguiller l'exception.

Nous montrons en figure 4.8 un exemple simpliste, il apparaît très clairement que la transparence n'était pas le but recherché.

Dans l'exemple, l'appel asynchrone est lancé en `1.` et le résultat est attendu en `2.`, l'exception potentielle aurait été obtenue en `3.` encapsulée dans une `ExecutionException`.

4.4.3 RAMI / Mandala

Le projet Mandala [15] comprend le logiciel RAMI [16] qui est une implémentation asynchrone de RMI. Plusieurs modèles d'asynchronisme sont proposés, ils sont plus ou moins transparents, et donc plus ou moins limités étant donnée la difficulté d'implémenter un mécanisme transparent.

L'approche par fonction de traitement est proposée pour gérer les exceptions, ainsi que la sauvegarde de l'exception dans le futur. Cette dernière est présentée différemment selon le niveau de transparence choisi. En mode non transparent, la méthode permettant

```
import java.util.concurrent.*;

public class JSR166 {
    private static String method(String s) throws InterruptedException {
       Thread.sleep(100); // throws InterruptedException
       System.out.println("Fin de la méthode");
       return s;
    }

    public static void main(String[] args) {
       ScheduledThreadPoolExecutor executor =
           new ScheduledThreadPoolExecutor(1);

       Future<String> f = executor.submit(new Callable<String>() { // 1.
           public String call() throws InterruptedException {
               return method("Hello World");
           }
       });

       executor.shutdown();
       System.out.println("Appel lancé");

       try {
           String resultat = f.get(); // 2.
           System.out.println("Résultat : " + resultat);
       } catch (Exception e) { // 3.
           System.out.println("Exception : " + e);
       }
    }
}
```

Figure 4.8. Exemple d'utilisation des futurs en Java 1.5

de récupérer le résultat a une déclaration `throws Throwable` dans sa signature, ainsi elle peut directement signaler une exception potentielle plutôt que de renvoyer un résultat. En mode transparent, le futur est simplement positionné à `null` en cas d'exception, ce qui provoquera une `NullPointerException` lors de l'accès. Ceci n'est hélas pas très précis pour aider le programmeur à comprendre l'erreur.

4.4.4 Conclusion sur l'exception dans le futur

L'avantage de cette approche est qu'elle permet de réutiliser la gestion d'exceptions du langage hôte de façon simple et non intrusive, c'est-à-dire que l'exception est traitée dans un bloc `catch` en Java. Par contre, lorsque l'asynchronisme est fourni de façon transparente, si l'exception est signalée loin de l'appel qui l'a produite, le système peut devenir incohérent. En effet, le bloc originellement prévu pour récupérer l'exception n'est plus

forcément encore en cours. De plus, aucun des mécanismes présentés n'implémente cette idée conjointement avec l'asynchronisme transparent. C'est pourquoi nous consacrons la section 5.3.2 au principal problème posé par une telle implémentation.

4.5 Autres approches

En dehors des deux principaux comportements de gestion d'erreur en asynchrone qui sont les fonctions de traitement et la sauvegarde de l'exception dans le futur, d'autres méthodes ont été imaginées.

4.5.1 Proxyc

Proxyc [17] est une autre implémentation d'objets futurs. Celle-ci fait appel à des modifications du bytecode au moment du chargement des classes. Leur solution au problème de la gestion d'exceptions est de rendre synchrone les appels susceptibles d'en signaler. Ils suggèrent plutôt d'encapsuler l'appel à exception dans une méthode sans exception qui s'occuperait donc de traiter cette exception. Cet appel servant d'encapsulation serait donc un appel asynchrone.

Ils proposent aussi une autre approche, utilisant leur modification du bytecode au chargement. Comme ils insèrent du code pour obtenir le véritable résultat à partir du futur, ils introduisent la possibilité de laisser le programmeur spécifier du code de gestion d'erreur à placer à cet endroit.

4.5.2 RMIX

RMIX [18] est une bibliothèque de communication en Java reprenant les concepts de RMI, mais pas l'implémentation. Elle a la caractéristique d'être multi-transport. Nous nous intéressons uniquement aux appels asynchrones, et leur gestion d'exceptions est assez originale. Chaque objet distant contient un booléen indiquant la présence d'une exception. Ce booléen est collant dans le sens où une fois positionné à vrai, il le reste jusqu'à qu'il soit explicitement remis à faux. Il est donc mis à vrai simplement suite à une exception dans un appel asynchrone, et peut être mis à faux de deux manières :

- en traitant une exception dans un appel synchrone,
- en utilisant manuellement l'API adéquate.

Quand le booléen est activé, l'objet interdit tout appel asynchrone.

4.6 Bilan

La première conclusion qui ressort de cet état de l'art est que l'approche par fonctions de traitement n'est pas du tout suffisante comme on pourrait le supposer. Ses deux principales limitations sont l'impossibilité de remonter la pile d'appel et l'absence de capture des variables locales.

En seconde conclusion, l'approche qui consiste à placer l'exception dans le futur semble intéressante mais délicate à réaliser. Elle est intéressante car elle permet de réutiliser les blocs `try/catch` mais est délicate à cause du risque de signaler une exception dans un mauvais bloc. Ceci peut arriver à cause d'une sortie anticipée du bloc pouvant récupérer une exception liée à un appel asynchrone.

5 Gestion d'exceptions en mode asynchrone

5.1 Vue d'ensemble

5.1.1 Principe

L'objectif est de fournir une gestion d'exceptions à la fois fonctionnelles et non-fonctionnelles en utilisant les `try/catch/finally` car ce sont les seuls moyens en Java de manipuler (remonter) la pile d'appels. De plus, avec un langage supportant les exceptions, la gestion d'erreur se doit d'être dans un bloc `catch`.

Pour se positionner par rapport à l'état de l'art, nous souhaitons fournir l'exception dans le futur. Nous souhaitons surtout assurer que l'exception soit signalée dans le même bloc `try` que l'appel originel, afin de conserver le comportement standard.

Le mécanisme doit être le plus transparent possible pour le programmeur, et respecter les contraintes de ProActive : pas de modification de bytecode ni de la JVM, pas de pré-traitement des sources.

Le but à atteindre est de pouvoir lancer un appel asynchrone même si la signature de la méthode appelée contient une déclaration `throws`. Un bloc `try/catch` est associé à cet appel, et afin de conserver le comportement habituel nous introduisons deux règles :

- lors de l'accès au futur l'exception potentielle est signalée,
- il est impossible de quitter le bloc `try` associé à l'appel avant son retour.

La solution proposée est donc de s'aider des délimitations (accolades) des blocs `try` pour proposer des appels asynchrones avec exceptions tout en gardant un comportement cohérent avec les appels synchrones. Ceci nécessite d'attendre avant de sortir d'un bloc `try` que tous les appels associés à ce bloc soient terminés.

L'utilisation du mécanisme consiste à annoter un bloc `try/catch/finally` aux endroits suivants : juste avant le `try`, à la fin du `try` et au début du `finally`.

L'unique but de ces appels ajoutés est d'indiquer à ProActive l'état de la pile des masques d'exceptions. Remarquons que ces instructions sont explicitement à fournir par le programmeur car Java ne donne aucune information sur cette pile des masques d'exceptions. Cette information n'étant intéressante que pour le client lors d'un appel asynchrone, très peu de données seront envoyées sur le réseau concernant ce mécanisme.

5.1.2 Survol rapide

Voyons maintenant un exemple afin d'avoir une vue globale du mécanisme proposé. Pour être activé, le mécanisme doit être appelé dans le code utilisateur à certains points clés montrés sur la figure 5.1. Les appels aux points clés sont :

1. `tryWithCatch()` doit être appelé juste avant le `try` et préciser quels types d'exceptions seront traités,
2. `endTryWithCatch()` doit être appelé à la fin du `try`,
3. nous verrons en section 5.2.1.4 la justification du `removeTryWithCatch()`.

```
public class Test {
    public DangerousThing dangerousMethod() throws DangerousException {
    }

    public static void main(String[] args) {
        ProActive.tryWithCatch(DangerousException.class); // (1)
        try {
            DangerousThing[] dt;
            Test[] t;

            for (int i = 0; i < dt.length; i++) {
                dt[i] = t[i].dangerousMethod();
            }

            ProActive.endTryWithCatch(); // (2)
        } catch (DangerousException de) {
            de.printStackTrace();
        } finally {
            ProActive.removeTryWithCatch(); // (3)
        }
    }
}
```

Figure 5.1. Exemple d'utilisation du mécanisme

5.2 Description du mécanisme

5.2.1 Cas général

Voyons maintenant en détail l'utilisation et le fonctionnement du mécanisme. Pour cela, nous allons explorer étape par étape le déroulement d'un bloc `try/catch` utilisant le mécanisme.

Avant le bloc `try`

Pour chaque `try` que l'on veut instrumenter avec le mécanisme, il faut spécifier quels types d'exceptions y sont traités, il n'y a aucun autre moyen de le savoir automatiquement en Java ; il faudrait avoir accès à l'exécution aux exceptions déclarées dans le `catch`. Ceci est réalisé grâce à la méthode statique `ProActive.tryWithCatch()` qui prend une classe ou un tableau de classes en paramètre.

Avec cette information, notre mécanisme d'exceptions asynchrones peut mettre à jour sa version de la pile de masques d'exceptions.

Remarquons que dans certains cas de `try/catch` imbriqués, le `tryWithCatch()` devient un point d'attente par nécessité. Ceci est exposé par l'exemple de la figure 5.2.

```
class ParentException extends Exception {}
class ChildException extends ParentException {}

ProActive.tryWithCatch(ChildException.class);
try {
    A a = ro.foo(); // throws ChildException

    /* Ici nous sommes obligés d'attendre le retour de ro.foo(). */
    ProActive.tryWithCatch(ParentException.class);
    try {
        a.bar();
        ProActive.endTryWithCatch();
    } catch (ParentException pe) { ... }
        ...
} catch (ChildException ce) { ... }
...
```

Figure 5.2. Cas où `tryWithCatch()` est un point d'attente par nécessité

Si dans ce code, l'appel `ro.foo()` génère une `ChildException`, on ne veut pas qu'elle soit signalée au moment de l'accès au futur « `a` » c'est-à-dire au moment de l'appel `a.bar()`. Si cela arrivait, l'exception serait traitée par le `catch (ParentException pe)` plutôt que par le `catch (ChildException ce)` qui lui était destiné, car `ChildException` est une sous-classe de `ParentException`. On est donc obligé d'attendre la fin de `ro.foo()` avant d'aller dans le `try/catch` de `a.bar()`, et cela sera réalisé par `ProActive.tryWithCatch(ParentException.class)`.

Il est important de remarquer que la situation est semblable dans le cas inverse. Si l'imbrication des deux blocs est inversée, c'est-à-dire si c'est le bloc pour `ParentException` qui contient celui pour `ChildException`, et si `ro.foo()` déclare `ParentException` la situation est différente mais le résultat est le même. En effet, l'exception de `ParentException` pourrait être en fait de type `ChildException`.

Pour résumer, dans le premier cas le problème est que le `catch (ParentException pe)` traitera forcément une `ChildException` et dans le second cas, le `catch (ChildException pe)` a la possibilité de traiter une `ParentException` si l'exception est aussi une `ChildException`. La raison est qu'une `ChildException` est une `ParentException`, et une `ParentException` peut être une `ChildException`.

Le rôle de cette primitive est donc le suivant :

1. attendre qu'il n'y ait plus d'appels en attente compatible avec la liste de types d'exceptions introduite,
2. empiler un niveau correspondant aux exceptions gérées,
3. recalculer le masque courant d'exceptions traitées.

Au sein du bloc

Ici, on s'attend à exécuter des appels asynchrones renvoyant des exceptions correspondant aux types annoncés.

Si un appel asynchrone génère une exception, elle est stockée dans le futur pour être lancée au moment de son accès.

Si l'appel de méthode est synchrone en raison de son type de retour (type primitif comme `int` ou une classe `final`) le comportement n'est pas changé : l'exception est lancée à la fin de l'appel qui est synchrone.

Dans le cas d'un appel one-way, le seul moment où l'on peut signaler l'exception est celui de l'appel de la méthode `ProActive.endTryWithCatch()`. Ceci pose problème car le programmeur est forcer à faire un compromis pour les appels one-way. D'un côté il a intérêt à espacer les `try` des `catch` pour que le `ProActive.endTryWithCatch()` arrive tard et ainsi permettre un parallélisme plus important, mais de l'autre il faudrait rapprocher les `try` des `catch` pour obtenir rapidement l'exception potentielle.

La solution proposée consiste en deux méthodes qui permettent de lancer une exception s'il y en a : l'une bloquante et l'autre non. La méthode non bloquante (`ProActive.throwArrivedException()`) se contente de consulter l'exception courante puis de la signaler si elle existe, tandis que la méthode bloquante (`ProActive.waitForPotentialException()`) agit comme une sorte de barrière. Elle attend la fin de tous les appels susceptibles de signaler immédiatement une exception, puis signale l'exception potentielle.

On comprend donc la différence de sens entre ces deux primitives. Celle non bloquante donne la possibilité de signaler une exception reçue, alors que la bloquante est utilisée pour s'assurer que tous les appels précédents ont réussi.

Il reste un cas à envisager. Imaginons qu'un futur contienne une exception, elle sera donc signalée lors du premier accès, mais que faire pour les accès suivants ? En Java normal (synchrone), ce problème ne se pose pas car si l'appel s'est soldé par une exception, l'affectation du résultat n'a pas eu lieu, et la variable garde donc son ancienne valeur. Dans

notre cas asynchrone, l'affectation est effectuée de toutes façons avant le retour de l'appel, avec le futur, et de plus, il est impossible d'être notifié de cette affectation, sauvegarder puis restaurer l'ancienne valeur n'est donc pas envisageable. Il reste donc deux possibilités pour traiter un tel accès :

- relancer l'exception contenue dans le futur,
- renvoyer `null`, ce qui provoquera a priori une `NullPointerException`.

La première solution est mauvaise puisque l'exception a déjà été lancée, ainsi l'exécution est sortie du `try/catch` associé, et donc en signalant l'exception une deuxième fois, elle s'arrêtera forcément dans un bloc `catch` inapproprié. La deuxième solution n'est pas non plus satisfaisante car `null` ne sera pas la valeur précédente de la variable d'après le raisonnement suivant : si le programmeur décide d'utiliser la valeur de la variable malgré le fait que la dernière affectation n'a pas pu être réalisée, c'est sûrement parce que la variable avait une valeur pertinente avant l'affectation, donc autre chose que `null`. Nous nous sommes quand même arrêtés sur cette dernière solution car elle semble moins perturbante que la première, mais il est clair que c'est un problème qu'il faut continuer d'explorer.

À la fin du bloc

La dernière instruction du bloc `try` doit être `ProActive.endTryWithCatch()`, le rôle de cette méthode est d'attendre le premier de l'un de ces deux événements :

- tous les appels du bloc susceptibles de générer une exception sont terminés,
- une exception est signalée.

Cette méthode sert aussi à « dépiler » le masque d'exceptions. Si une exception est signalée avant cet appel, il ne sera pas exécuté car le flot de contrôle aura quitté le bloc. Nous verrons en section 5.3.3 comment cette situation est gérée.

Après le bloc

Rappelons que nous sommes obligés de construire notre propre pile de masques d'exceptions car celle de Java est impénétrable. Java fournit uniquement la pile d'appels grâce à une méthode se trouvant, ironiquement, dans la classe `Throwable` (`getStackTrace()`). Ceci est vu comme un défaut de Java, sauf si le but est de cacher le fonctionnement de son système d'exceptions afin de l'optimiser [19] au niveau du JIT.

Le soin de conserver les deux piles (celle de Java et la nôtre) synchronisées à l'entrée et à la sortie d'un bloc `try` est laissé aux appels `ProActive.tryWithCatch()` et `ProActive.endTryWithCatch()`. L'imbrication de ces appels représente notre pile. Hors, le principe des exceptions est d'avoir des flots d'exécution court-circuitant les méthodes, il est donc prévisible et même attendu que des blocs `try` soient quittés sans passer par l'appel `ProActive.endTryWithCatch()` à la fin du bloc. Il faut aussi garder à l'esprit que la pile que nous construisons n'est pas complète puisqu'elle n'inclut que les blocs `try/catch` du code utilisateur manipulant des objets actifs ProActive.

Nous ne pouvons pas être prévenus lorsqu'une exception est signalée, donc nous ne pouvons pas prévoir les déroutements de flot de contrôle. Même pour les exceptions lancées par notre mécanisme, nous ne pouvons pas savoir exactement où elles seront traitées étant donnée que nous ne connaissons qu'un sous-ensemble de la pile d'exceptions.

La solution pour conserver une pile cohérente repose sur le mécanisme `finally` qui garantit que le code placé dans ce bloc sera exécuté. Nous y plaçons donc le code pour dépiler notre version de la pile d'exceptions. Ainsi, n'importe quelle exception remontant un certain nombre de blocs déclenchera les blocs `finally` sur son passage qui mettront à jour la pile des masques d'exceptions.

Le bloc `finally` devient donc obligatoire et doit commencer par un appel à la méthode `ProActive.removeTryWithCatch()`. Cette méthode a pour but de mettre à jour notre pile de masques d'exceptions.

5.2.2 Cas particuliers

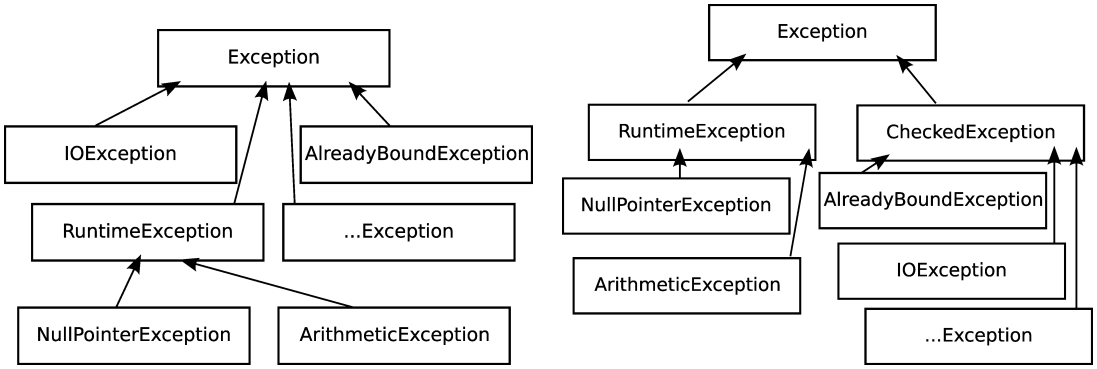
Exceptions non annoncées

Pour chaque appel avec exceptions, le mécanisme présenté consulte la déclaration `throws` pour savoir s'il est possible de lancer l'appel en asynchrone. En effet, il est nécessaire que tous les types d'exceptions annoncés dans le `throws` soient connus par le masque courant du mécanisme afin de garantir que l'exception soit gérée dans un `catch` cohérent.

Cependant il est possible de signaler des exceptions sans les avoir annoncées, c'est le cas en Java avec les `RuntimeException`. Ces exceptions sont potentiellement signalées par tous les appels, ce qui implique qu'en l'absence de traitement spécifique il faudrait traiter tous les appels en synchrones. La solution retenue sans le mécanisme n'était pas celle là mais, rappelons-le, était de signaler l'exception lors de l'accès au futur sans se soucier de la présence d'un bloc `catch` correspondant. Pour les appels one-way, le comportement proposé était d'afficher une trace du côté serveur.

Grâce aux annotations imposées par le mécanisme présenté, il devient possible de connaître les endroits où les `RuntimeException` sont traitées. Le programmeur pourra donc indiquer, tout comme pour les exceptions normales, les limites des blocs les traitant. Étant donné que tout appel est susceptible de générer n'importe quelle `RuntimeException` tous les appels au sein d'un bloc traitant un sous-type quelconque de `RuntimeException` seront considérés comme compatibles au niveau des exceptions, et donc le flot de contrôle ne sortira pas du bloc avant le retour de tous ces appels.

Il y a quand même un inconvénient dans cette utilisation, lié à la conception des classes d'exceptions en Java. La classe mère de toutes les exceptions est la classe `Exception`, elle a comme sous-classes directes la classe `RuntimeException` ainsi qu'un nombre très important d'autres exceptions qui ne sont pas des `RuntimeException`.



Hiérarchie réelle Hiérarchie préférable

Figure 5.3. Hiérarchies de classes d'exceptions

Comme le montre l'extrait de la hiérarchie des exceptions en figure 5.3, la classe `RuntimeException` est une sous-classe comme une autre alors qu'elle possède une différence fondamentale dans sa gestion des `throws`. Il aurait été préférable d'introduire une sous-classe, que nous nommons `CheckedException`, comme super-classe de toutes les exceptions autres que les `RuntimeException`. On rencontre souvent des `catch (Exception e)` dans du code Java dans le but de traiter tous les exceptions déclarées, il aurait été souhaitable de pouvoir faire `catch (CheckedException ce)` dans ce cas afin de mieux exprimer l'intention. La conséquence de cette situation sur notre mécanisme est qu'un `catch (Exception e)` agira comme une barrière pour tous les appels du bloc car il inclut les `RuntimeException` potentiellement signalées par n'importe quel appel.

Exceptions non-fonctionnelles

Les exceptions non-fonctionnelles sont générées par le middleware plutôt que par le code utilisateur, ainsi nous avons une plus grande liberté pour choisir une gestion.

ProActive dispose déjà d'une stratégie de gestion de ces exceptions utilisant des fonctions de traitement, appelée NFE. Ayant vu les limitations de ce genre de mécanisme, il paraît intéressant de pouvoir, si on le souhaite, gérer ces exceptions non-fonctionnelles de la même manière que les exceptions normales (fonctionnelles), c'est-à-dire en utilisant la construction `try/catch` et l'API proposée.

Nous souhaitons donc nous rapprocher de RMI avec ses `RemoteException`, en évitant d'introduire son inconvénient : son aspect obligatoire et donc sa tentation de placer des `catch (RemoteException re)` vides afin d'éviter les erreurs à la compilation. Nous voulons donc laisser au programmeur la possibilité d'ignorer ces exceptions dans le but de les gérer avec les fonctions de traitement. Il s'agit donc de remonter ces exceptions au programmeur uniquement lorsqu'il les demande dans un bloc `catch`.

Une fois de plus, le système d'exceptions de Java complique la situation, même si cette fois c'est une de ses qualités qui intervient. Résumons la situation : nous ne voulons pas ajouter des exceptions aux déclarations `throws` de toutes les méthodes pour ne pas tomber

dans le cas RMI, mais le programmeur doit pouvoir placer des blocs `catch` pour ces exceptions. Le compilateur Java signalera alors une erreur : les blocs `catch` sont inaccessibles car aucun appel ne déclare signaler les exceptions traitées. La solution retenue consiste à placer ces exceptions non-fonctionnelles comme sous-classe des `RuntimeException`, ainsi le compilateur saura que ces exceptions peuvent être signalées à chaque appel et nous nous retrouvons dans le cas de la section précédente.

Il est possible d'avoir un comportement encore meilleur sur les exceptions non-fonctionnelles, car elles sont signalées uniquement par le middleware. Par exemple, certaines exceptions ne peuvent être déclenchées qu'à des instants précis bien connus du middleware, comme l'impossibilité d'envoyer la requête pour exécuter un appel. Une fois que la requête est envoyée, si l'appelant traite uniquement cette exception il est inutile d'attendre le retour de l'appel puisque l'exception ne pourra plus être signalée. Mais ceci est encore au stade de l'étude.

Exceptions consécutives

Implémenter un mécanisme d'exceptions asynchrones en utilisant les exceptions de Java tout en fournissant les exceptions au même niveau que celles de Java peut conduire à des situations problématiques. Dans le monde des exceptions normales (synchrones), quand un appel se termine par une exception, le flot de contrôle remonte la pile, et donc les appels suivants dans les blocs traversés ne sont pas exécutés.

Dans le monde asynchrone, le but est de ne pas attendre la fin des appels, il est donc tout à fait possible que deux appels soient exécutés en parallèle et se terminent tous les deux par une exception. Normalement dans ce cas, les exceptions devraient avoir la même cause : une panne réseau par exemple. Nous nous retrouvons donc avec deux exceptions à remonter au programme.

Dans le cas synchrone ceci n'est pas possible car si le premier appel échoue, le deuxième n'est pas lancé, c'est-à-dire que seule la première exception est retournée. Nous allons appliquer cette sémantique en ne gardant que la première exception, et donc en ignorant les exceptions suivantes tant que la première n'a pas été signalée dans le programme.

Il reste le problème que le deuxième appel est lancé à tord étant donné l'échec de son précédent. Pour cela nous nous en remettons aux accès au futurs, potentiellement des attentes par nécessité. En effet, ces accès créent un graphe de dépendance entre les valeurs des futurs. Nous émettons donc la supposition que le graphe de dépendance des valeurs de retour est le même que le celui des appels de méthodes. Ceci semble naturel.

Une autre approche a été considérée. Cette approche consiste à signaler l'exception asynchrone le plus rapidement possible. À chaque appel dans la librairie ProActive que ce soit implicitement à cause d'un appel asynchrone, ou explicitement en utilisant l'API de ProActive, l'exception courante peut-être consultée et signalée si présente. Ceci ne pose pas de difficulté technique, mais peut être troublant pour le programmeur car les exceptions seraient signalées par des appels sans rapport.

Cette dernière solution aurait l'avantage de rendre le mécanisme proposé plus proche que le mécanisme normal de Java, comme on peut le voir dans l'exemple de la figure 5.4. Dans cet exemple, quatre appels sont lancés en parallèle.

```
ProActive.tryWithCatch(FooException.class);
try {
    Result r1 = ro1.foo1(); // throws FooException
    Result r2 = ro1.foo2(); // throws FooException
    Result r3 = ro2.foo1(); // throws FooException
    Result r4 = ro2.foo2(); // throws FooException
    ProActive.endTryWithCatch();
} catch (FooException fe) {
    ...
} finally {
    ProActive.removeTryWithCatch();
}
```

Figure 5.4. Exemple où signaler l'exception le plus tôt possible semble être utile

On s'attendrait à sortir du `try` dès qu'une exception est lancée, mais tel que prévu il faudrait attendre le `ProActive.endTryWithCatch()` pour avoir une exception. Avec la modification imaginée, si une exception arrive avant que tous les appels soient démarrés, elle serait signalée lors du prochain appel qui n'aurait donc pas lieu ainsi que ses successeurs. Malheureusement, cette dépendance sur la durée de lancement des appels peut rendre le mécanisme imprévisible. C'est pourquoi, cette approche est laissée de côté en attendant d'avoir plus d'expérience sur le sujet.

5.3 Implémentations

5.3.1 Structure de données

Le mécanisme ayant été implémenté, intéressons-nous à la structure de données utilisée pour représenter son état. La figure 5.5 en montre une instance.

Détaillons les éléments la composant :

- un état courant permettant d'éviter de consulter la pile pour la plupart des manipulations :
  - l'exception reçue est sauvegardée afin d'être envoyée quand elle sera demandée,
  - l'union des masques de tous les niveaux de la pile est utilisé pour savoir rapidement si un type d'exception est géré par le mécanisme ou pas,
- une pile matérialise l'imbrication des différents blocs `try` et comprend pour chaque niveau :
  - le nombre d'appels asynchrones encore en attente afin de savoir quand il est possible de quitter le bloc `try`, c'est-à-dire lorsque tous les futurs sont revenus,
  - le masque d'exceptions géré par ce bloc `try`.

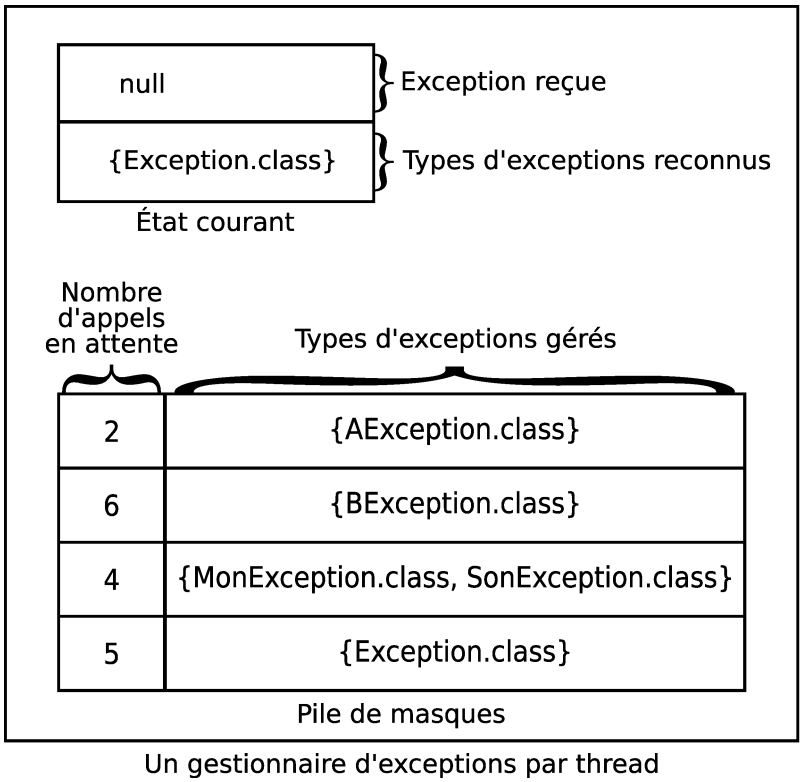


Figure 5.5. Schéma de la structure de données

Remarquons aussi qu'il existe une unique instance de cette structure de donnée pour chaque thread. En effet, la pile de masques d'exceptions étant liée à la pile d'appel, elle est spécifique à chaque thread.

5.3.2 Signalisation d'une exception

Pour implémenter ce mécanisme d'exceptions il est nécessaire de pouvoir signaler des exceptions à partir de n'importe quel endroit dans le code. En effet, cette gestion est réalisée au-dessus de celle de Java mais surtout à sa place, c'est donc notre mécanisme qui s'occupe de signaler les exceptions.

En fonction du langage de programmation hôte, cette possibilité n'est pas toujours offerte. En ce qui nous concerne, elle ne l'est pas réellement en Java. En effet, le mécanisme des déclarations `throws` vérifiées à la compilation empêche de signaler une exception à partir de n'importe quel endroit. Nous allons donc voir comment contourner ce mécanisme en Java.

Le premier point à considérer est que cette règle du `throws` est appliquée par le compilateur, c'est-à-dire qu'il suffit de ne pas passer par le compilateur pour atteindre notre but. Rappelons aussi qu'une contrainte de ProActive est de ne modifier ni le compilateur ni la JVM.

La méthode retenue pour contourner le compilateur est de dynamiquement créer le bytecode à l'exécution avec l'aide d'une bibliothèque comme ASM [20]. Cette bibliothèque nous permet de créer dynamiquement une méthode qui signale l'exception passée en paramètre tout en ayant une déclaration `throws` vide. Il est donc possible de générer dynamiquement une telle classe, puis de la charger en appelant la méthode protégée `defineClass()` par réflexion.

Maintenant que nous avons notre méthode pour signaler n'importe quelle exception, il faut encore pouvoir l'appeler. Nous ne pouvons pas simplement l'appeler par son nom, comme elle est créée dynamiquement, elle n'existe pas à la compilation, ce qui provoquerait une erreur. Appeler cette méthode via l'API de réflexion ne convient pas non plus car la méthode `invoke()` servant à exécuter une méthode par réflexion a une déclaration `throws` non vide.

La solution complémentaire est d'utiliser la liaison dynamique de Java pour appeler cette méthode. Nous créons donc une petite interface comme sur la figure 5.6 qui sera notre point d'entrée pour signaler une exception. Remarquons que la méthode `throwException()` n'a pas de déclaration `throws`.

```
public interface Thrower {
    public void throwException(Throwable t);
}
```

Figure 5.6. Interface pour signaler une exception

La classe construite dynamiquement doit donc implémenter cette interface, et ainsi il suffit de construire par réflexion une instance de cette classe et de l'affecter à une variable du type de l'interface. En appelant la méthode `throwException()` sur cette instance, par liaison dynamique ce sera celle de la classe construite dynamiquement qui sera appelée. Elle va donc, à notre demande, signaler l'exception passée en paramètre.

5.3.3 Modification du code source

Java ne fournissant pas d'interface pour consulter/manipuler le masque des exceptions traitées, nous sommes obligés d'ajouter du code pour effectuer ces manipulations aux moments opportuns. Le code ajouté est redondant par rapport au `try/catch` mais est nécessaire. Le fait que ce code soit redondant est à la fois un avantage et un inconvénient. L'avantage est que cette modification peut être automatisée, mais l'inconvénient est le risque de désynchronisation avec le véritable `try/catch` lorsque le programmeur écrit lui-même le code ajouté.

Concernant la modification automatique du code source, il ne s'agit pas d'un prétraitement ou d'une phase de pré-compilation. L'utilisateur voulant utiliser ce mécanisme d'exceptions sait qu'il doit ajouter les appels vus précédemment, il sait aussi que cet ajout est totalement mécanique une fois les blocs `try/catch` sélectionnés. Il s'agit donc de faire un programme qui ajoute ces appels sur des sources Java. La solution retenue pour ce travail est un analyseur lexical couplé avec un analyseur syntaxique, c'est-à-dire le couple lex/yacc adapté à Java.

Construire l'arbre syntaxique complet pour un programme Java est inutile, et problématique compte tenu des changements occasionnels dans la grammaire Java. Nous nous contentons donc uniquement de représenter les blocs (entre accolades) et les annotations ajoutées. Considérer tous les blocs plutôt qu'uniquement les constructions permet de grandement simplifier la grammaire, et inclure les annotations que nous ajoutons permet de les effacer. L'intérêt d'effacer nos annotations est d'avoir une transformation idempotente.

La modification automatique du code source consiste donc en ces étapes :

- supprimer les appels de méthodes ajoutés automatiquement,
- ajouter le `ProActive.tryWithCatch()` avec les bons paramètres juste avant le `try`,
- ajouter le `ProActive.endTryWithCatch()` à la fin du `try`,
- ajouter le `ProActive.removeTryWithCatch()` au début du `finally` en construisant ce bloc s'il n'existe pas déjà.

6 Exceptions non-fonctionnelles

Comme nous l'avons vu précédemment, la bibliothèque ProActive maintient la distinction entre les exceptions fonctionnelles et non-fonctionnelles. Rappelons ici la différence, elle est basée de la notion de proprietés non-fonctionnelles introduite par les composants. Les exceptions fonctionnelles concernent le code applicatif utilisant la bibliothèque ProActive pour la partie distribution, c'est-à-dire qu'il est responsable de la création et de l'utilisation de ces exceptions. Le chapitre précédent expliquait la gestion de ces exceptions dans l'environnement asynchrone fourni par ProActive.

Quant aux exceptions non-fonctionnelles elles appartiennent au middleware, dans notre cas c'est donc ProActive qui les crée et qui les traîte, quitte à ce que le traitement soit délégué au code applicatif.

6.1 Spécification

La gestion des exceptions non-fonctionnelles dans ProActive est définie dans [6], rappelons-en brièvement les principes, que nous avons déjà abordés en section 4.3.1. Elle commence par définir une hiérarchie d'exceptions présentée en figure 6.1, qui pourra être remodelée au fur et à mesure de l'implémentation d'autres fonctionnalités.

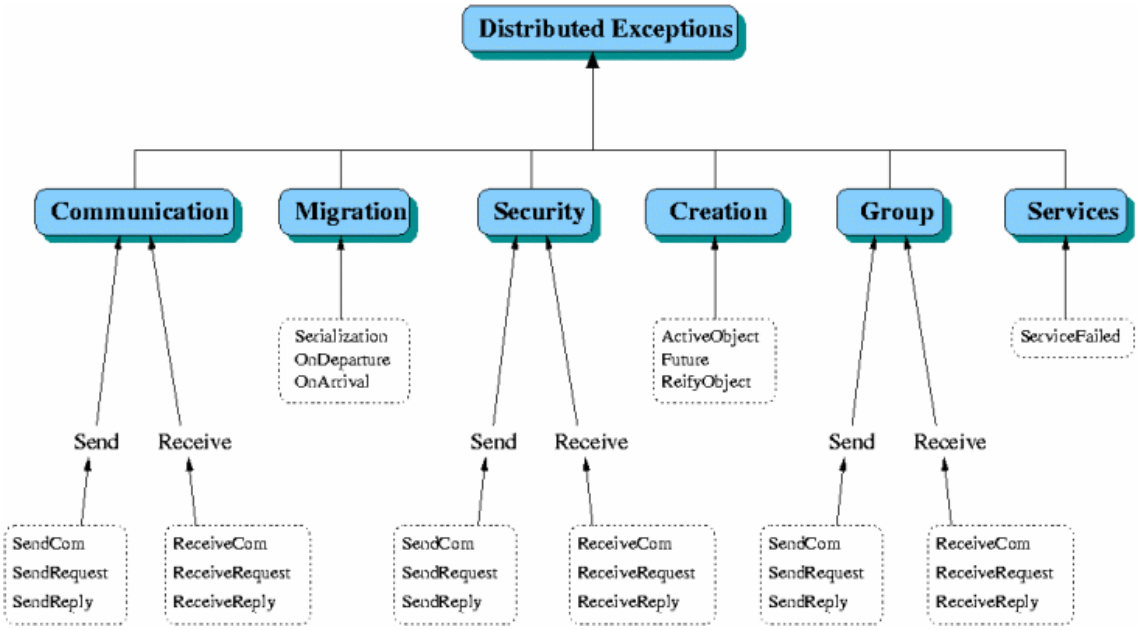


Figure 6.1. Hiérarchie d'exceptions non-fonctionnelles

Comme nous l'avons vu en section 4.3 le principe est d'associer des fonctions à des types d'exceptions et autres critères. Quand une exception survient, la fonction appropriée est recherchée puis exécutée.

La spécification originelle prévoyait les critères suivants pour le choix de la fonction de traitement :

- le type de l'exception (sa classe),
- la jvm (c'est-à-dire le processus exécutant l'application),
- l'objet actif concerné,
- la vision locale de l'object actif,
- le futur (résultat en attente),
- le cas par défaut.

Nous l'avons simplifié en supprimant les critères suivants :

- la jvm (c'est-à-dire le processus exécutant l'application) :
  - il suffit de mettre une fonction par défaut sur la jvm correspondante,
- le futur (résultat en attente) :
  - les opérations sur le futur en attente sont sujettes à des problèmes de concurrence car au moment de l'opération on ne peut pas se synchroniser sur la méthode à l'origine de la création du futur.

Ainsi, le comportement prévu en cas d'exception est de sélectionner la liste des traitements associée aux conditions de l'exception. Pour chacun de ces traitements on l'interroge pour savoir s'il est concernée par l'exception courante (une fonction est prévue à cet effet) auquel cas sa fonction de traitement est appelée.

Le comportement par défaut quand aucune fonction n'est trouvée pour une exception est de conserver une trace écrite de cette exception dans les messages d'erreur de l'application.

6.2 Implémentation

L'implémentation de ce mécanisme de gestion d'exceptions non-fonctionnelles a été refaite. Elle se base maintenant sur le modèle des « listeners » de Java. Ainsi les éléments susceptibles de générer des exceptions non-fonctionnelles contiennent une liste de fonctions de traitement et exportent des méthodes pour manipuler cette liste.

7 Application

Nous venons donc de présenter un mécanisme permettant d'effectuer des appels de méthodes en asynchrone avec possibilité de signalisation et récupération des exceptions. Afin de compléter la présentation, examinons un exemple d'application. Nous verrons donc les possibilités offertes par ce mécanisme.

7.1 Arithmétique robuste

L'exemple que nous avons retenu concerne les calculs d'arithmétique utilisant les fractions. Nous ne prétendons pas réaliser un système de calcul distribué complet, c'est pourquoi ses capacités distribuées sont très limitées. L'application permet de distribuer le calcul d'une somme sur plusieurs machines. Plus précisément, il est possible de calculer des formules du type $\sum_{n=a}^{b} f(n)$ en précisant a, b et f. La formule $f(n)$ est représentée par une interface contenant une seule fonction. Cette fonction définie par l'utilisateur lors de l'implémentation de l'interface précisera donc les termes de la somme à évaluer.

Voyons maintenant le côté robuste de cet exemple. Il réside dans la signature de la fonction à implémenter : `public Ratio eval(int x) throws OverflowException;` et, comme on le voit, la fonction est susceptible de signaler des exceptions. Ces exceptions sont utilisées pour prévenir des débordements de capacité (overflow). Toutes les méthodes de calcul ont une clause `throws OverflowException` et la vérification en elle-même est effectuée dans la couche la plus basse. L'apparition d'overflow dans un calcul est une erreur qui peut arriver à tout moment, c'est donc un exercice intéressant pour le mécanisme.

Pour obtenir des overflows en Java, nous ne travaillons pas avec les types primitifs (`int`, `long`) trop réduits, mais utilisons les `BigInteger`. C'est une classe fournie dans la bibliothèque standard Java permettant de manipuler des nombres sans bornes au prix d'une consommation mémoire accrue. En introduisant les overflows, nous pouvons donc limiter cette consommation mémoire.

7.2 Approximation de π

Il s'agit maintenant de trouver un exemple pour le programme d'arithmétique robuste. Nous nous sommes arrêtés sur la formule [21] de Fabrice Bellard montrée en 7.1 permettant de calculer π. Cette formule a l'avantage de ne mettre en œuvre que des nombres rationnels, ainsi les calculs sont exacts.

$$\frac{1}{2^6}\sum_{n=0}^{\infty}\frac{(-1)^n}{2^{10n}}\left(-\frac{2^5}{4n+1}-\frac{1}{4n+3}+\frac{2^8}{10n+1}-\frac{2^6}{10n+3}-\frac{2^2}{10n+5}-\frac{2^2}{10n+7}+\frac{1}{10n+9}\right) \quad (7.1)$$

Le programme d'exemple demande donc deux paramètres pour lancer le calcul. Ces deux paramètres sont :

- la borne supérieure, ce qui donne la précision du calcul,
- la taille maximum des nombres, pour savoir quand signaler un overflow.

Avec ces deux paramètres, nous pouvons lancer le calcul, ce qui donnera au final une fraction approximant π, ou une exception si un overflow est survenu.

Comparer précisément les performances de l'exemple avec et sans le mécanisme d'exceptions est spectaculaire grâce à la différence fondamentale apportée par le mécanisme. En effet, en désactivant le mécanisme d'exceptions asynchrones, les appels sont synchrones car ils déclarent des exceptions. L'exécution est répartie sur plusieurs machines, mais si les appels sont synchrones, la somme est quand même calculée séquentiellement, anéantissant les bénéfices de la distribution. Au contraire, avec le mécanisme d'exceptions asynchrones activé, les appels devenant asynchrones ils sont exécutés en parallèle sur les différentes machines. Ainsi, la somme est découpée en sous-sommes, et celles-ci sont évaluées en parallèle.

En testant un calcul de 100 itérations sur quatre machines, nous passons de 60 secondes à 20 secondes, ce qui indique une vitesse triplée. Même dans un test réduit comme celui-ci, une faiblesse de l'exemple apparaît : la somme est découpée en donnant autant de termes à calculer à chaque machines. Ce découpage se base sur l'idée que le calcul de chaque terme a la même durée, ce qui est faux. Les termes les plus grands prennent plus de temps à être évalués, mais de toutes façons ce n'était pas l'objet du mécanisme de résoudre ces problèmes de distribution.

8 Conclusion et perspectives

Nous avons exploré le problème de la gestion des exceptions en environnement distribué, avec une sémantique d'asynchronisme transparent. En explorant les solutions existantes, deux approches se sont démarquées : les fonctions de traitement, et le placement de l'exception dans le futur.

L'étude des solutions utilisant les fonctions de traitement a montré deux limitations : le contexte capturé n'est pas suffisant, et il est impossible de remonter la pile. Ces deux limitations n'existent pas dans le mécanisme habituel des exceptions.

L'autre approche majeure, l'exception dans le futur, ne possède pas ces limitations car elle réutilise le mécanisme d'exception fourni par le langage. Cependant, la gestion d'exception fournie par le langage est très dépendante de l'aspect synchrone des appels de méthodes. C'est pourquoi la mise en œuvre de cette approche est délicate.

La solution proposée dans ce mémoire est de réutiliser les délimitations des blocs utilisés avec les exceptions. Ces blocs deviennent alors des barrières pour les appels asynchrones, mais spécifiquement pour les exceptions qui peuvent être traitées dans chaque bloc. Cette solution permet à la fois de traiter les exceptions fonctionnelles et de fournir un comportement adéquat aux exceptions non-fonctionnelles.

Remarquons qu'imposer une barrière à la fin d'un bloc ne semble pas problématique en suivant ce raisonnement : l'intérêt de ces blocs de code est de pouvoir bien séparer le code « normal » de celui de gestion d'erreur. Les blocs de code « normal » ne contiennent donc aucune gestion d'erreur. Ainsi, généralement ces blocs contiennent à la fois l'exécution d'un appel et l'utilisation de son résultat. C'est donc cette utilisation de résultat qui provoquera l'attente sur le futur, et non la barrière à la fin du bloc.

Les incompatibilités entre l'asynchronisme et les systèmes de gestion d'exceptions n'ont pour autant pas toutes été levées. Nous envisageons donc comme perspectives l'étude des problèmes suivants :

- utilisations multiples d'un futur qui a déjà signalé une exception,
- stratégie pour signaler une exception le plus tôt possible pour empêcher les exceptions consécutives tout en conservant un comportement prévisible,
- certains blocs imbriqués nécessitent des barrières même à leur entrée, d'où une limitation potentiellement inutile de l'asynchronisme.

Nous pensons que le travail commencé ici donne des éléments pour aborder l'étude de ces problèmes restants et généraliser l'approche. Il sera intéressant d'appliquer ce mécanisme aux communications de groupes, où un seul appel peut provoquer la signalisation de plusieurs exceptions. Cette approche pourra aussi fournir une gestion d'exceptions aux composants asynchrones.

Bibliographie

[1] Caromel, D. and Chazarain, G. (2005). Robust exception handling in an asynchronous environment. In Romanovsky, A., Dony, C., Knudsen, J. and Tripathi, A., editors, *Developing Systems that Handle Exceptions. Proceedings of ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems. Technical Report No 05-050*. Department of Computer Science. LIRMM. Montpellier-II University. 2005. July. France.

[2] Goodenough, J. B. (1975). Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696.

[3] Forum, M. P. I. (1994). MPI: A message passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):159–416.

[4] Caromel, D., Klauser, W. and Vayssière, J. (1998). Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061.

[5] Wollrath, A., Riggs, R. and Waldo, J. (1996). A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems (COOTS)*, pages 219–232. USENIX Association.

[6] Caromel, D. and Genoud, A. (2003). Non-functional exceptions for distributed and mobile objects. In Romanovsky, A., Dony, C., Knudsen, J. and Tripathi, A., editors, *Proceedings of the Exception Handling in Object-Oriented Systems workshop at ECOOP 2003*.

[7] Keen, A. W., Ge, T., Maris, J. T. and Olsson, R. A. (2004). Jr: Flexible distributed programming in an extended java. *ACM Trans. Program. Lang. Syst.*, 26(3):578–608.

[8] Olsen, R. A., Andrews, G. R., Coffin, M. H. and Townsend, G. M. (1992). Sr: A language for parallel and distributed programming. Technical Report, University of Arizona.

[9] Keen, A. W. and Olsson, R. A. (2002). Exception handling during asynchronous method invocation (research note). In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 656–660, London, UK. Springer-Verlag.

[10] Welch, I. and Stroud, R. J. (2000). Kava - a reflective java based on bytecode rewriting. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 155–167, London, UK. Springer-Verlag.

[11] Welch, I., Stroud, R. J. and Romanovsky, A. B. (2001). Aspects of exceptions at the meta-level. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 280–281, London, UK. Springer-Verlag.

[12] Raje, R. R., Williams, J. I. and Boyles, M. (1997). Asynchronous Remote Method Invocation (ARMI) mechanism for Java. *j-CPE*, 9(11):1207–1211.

[13] Logan, P. (Cunningham & Cunningham Wiki). Closures that work around final limitation. . http://c2.com/cgi/wiki?ClosuresThatWorkAroundFinalLimitation.

[14] Lea, D. (2002–2004). Java specification requests 166: Concurrency utilities. . http://www.jcp.org/en/jsr/detail?id=166.

[15] Vignéras, P. (2004). *Vers une programmation locale et distribuée unifiée au travers de l'utilisation de conteneurs actifs et de références asynchrones.*. PhD thesis, Université de Bordeaux 1, LaBRI.

[16] Vignéras, P. (2005). Mandala/rami user's guide. . http://mandala.sf.net/docs/rami.pdf.

[17] Pratikakis, P., Spacco, J. and Hicks, M. (2004). Transparent proxies for java futures. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 206–223, New York, NY, USA. ACM Press.

[18] Kurzyniec, D. and Sunderam, V. S. (2004). Semantic aspects of asynchronous RMI: The RMIX approach. In *Proc. of 6th International Workshop on Java for Parallel and Distributed Computing, in conjunction with IPDPS 2004*. IEEE Computer Society.

[19] Suganuma, T., Yasue, T. and Nakatani, T. (2003). A region-based compilation technique for a java just-in-time compiler. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 312–323, New York, NY, USA. ACM Press.

[20] Bruneton, E., Lenglet, R. and Coupaye, T. (2002). Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*. http://asm.objectweb.org/current/asm-eng.pdf.

[21] Bellard, F..A new formula to compute the n'th binary digit of pi. . http://fabrice.bellard.free.fr/pi/pi_bin.ps.

Résumé

Le présent mémoire rapporte le travail effectué lors de mon stage de recherche dans le projet OASIS à l'INRIA Sophia-Antipolis durant les mois de mars à septembre 2005.

Le but de ce travail de recherche était d'adapter le mécanisme des exceptions à une sémantique d'appels distants asynchrones.

Sommairement, la solution retenue consiste à permettre de l'asynchronisme au sein des blocs définis pour le mécanisme des exceptions mais d'enchaîner ces blocs de manière synchrone.